Design methodology for dynamically reconfigurable systems

Florent Berthelot , Fabienne Nouvel , Dominique Houzet
CNRS UMR 6164 IETR/INSA
Rennes 20 av des Buttes de Coesmes, 35043 Rennes, France
Email: florent.berthelot@ens.insa-rennes.fr
{fabienne.nouvel ; dominique.houzet}@insa-rennes.fr

***Abstract*— In this paper we present an automatic design generation methodology for heterogeneous architectures composed of microprocessors, DSPs and FPGAs. This methodology is based on an adequation algorithm architecture where application is represented by a control data flow graph and architecture by an architecture graph. We focus on how to take into account specificities of partially reconfigurable components during the adequation process and for the design generation. We present a method which generates automatically the design for both fixed and partially reconfigurable parts of a FPGA. This method uses prefetching technic to minimize reconfiguration latency of runtime reconfiguration and buffer merging to minimize memory requirements of the generated design.**

I. Introduction

Recent developments in radio technology have introduced software defined radio paradigm SDR [1]. This evolution has emerged with the proliferation of wireless standards, including both local area networks 802.11 to 2,5G, 3G, and future 4G telecommunication standards. Future telecommunication systems will be software reprogrammable radios, which could be reconfigured to adapt for changing communication protocols and channels.

Such systems require heterogeneous architectures based on digital signal processors (DSPs), general purpose processors and reconfigurable devices like field programmable gate arrays (FPGAs). The whole application is split between HW/SW components during the partitioning process. This step leads to a compromise between system's performance of an hardwired solution and flexibility of a software solution.

These components use different levels of reconfigurations. Harvard-based architectures (DSPs, general purpose processors) are reconfigurable at system-level and use a temporal scheme implementation of operations. Configuration of the data path requires few data and can be performed at each cycle. This high flexibility is well adapted to control based applications but suffers from its power efficiency for repetitive and high throughput computations. The introduction of Dynamically Reconfigurable Systems (DRS), can deliver higher levels of performance, flexibility and power efficiency. Reconfigurable devices, including FPGAs, can fill the gap between hardwired and software technology. Recently runtime reconfiguration (RTR) of partial FPGA parts has led to the concept of virtual hardware. Its allows to change only a specified part of the chip while other areas remain operational and unaffected by the reconfiguration [2]. So RTR allows more sections of an application to be mapped into hardware, a larger part of the application can be accelerated by contrast with a microprocessor computation. By changing dynamically the functionality performed by the FPGA over the time, we can address SDR constraints and obtain a scalable system which can evolved in agreement with its environment requirements, and hence using a reconfigurable hardware platform across multiple standards. However reconfiguration latency is a major drawback of runtime reconfiguration on commercial devices and must be considered during HW/SW partitioning process. The objective is to obtain a near optimal scheduling of tasks in time over an heterogeneous architecture [3]. These more and more complex systems must be handled by an appropriate design flow to reduce development time under strong time-to-market pressure. These steps can be achieved by modeling application operations through control data flow graph (DFG) and operate an Adequation Algorithm Architecture (AAA) methodology.

In this paper we focus on codesign systems with DSPs and commercial FPGAs for telecommunication applications. The first section describes the tool SynDEx [4] used for the application partitioning stage. We describe the impact of run-time reconfigurable component utilization on algorithm-architecture adequation in the second section. Next modeling a partially runtime reconfigurable part of a FPGA with this tool is exposed. Then we present the automatic VHDL design generation for a runtime reconfigurable component. We discuss how to generate an automatic management of runtime reconfiguration over the time with SynDEx. As a proof an implementation example using a partially reconfigurable chip is presented in the last part of the paper.

II. AAA approach and SynDEx representation

Some partitioning methodologies based on various approaches are reported in the literature [5]. They are characterized by the granularity level of the partitioning, metrics, target hardware, support of runtime-reconfiguration, flow automation and on-line / off-line scheduling policies. Few tools, based on these partitioning methodologies, provide a seamless flow from the specification to the implementation.

Choice of candidates for a dynamic hardware implementation can be guided by some metrics: execution time, memory

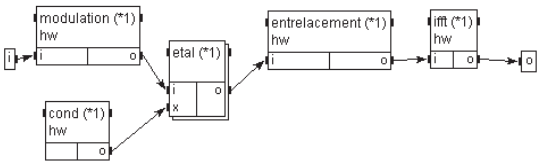


Fig. 1. Algorithm graph

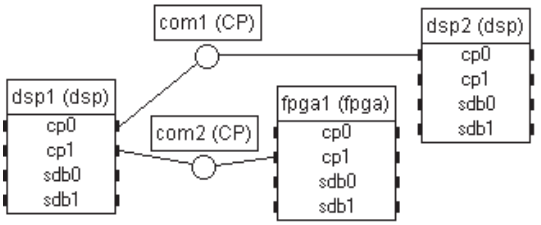


Fig. 2. Architecture graph

constraints, power efficiency, reconfiguration time, configuration prefetching capabilities and area constraints.

Among theses methods we have chosen SynDEx already used in [6]. Many libraries have been developed for heterogeneous platforms.

SynDEx is an academic system-level CAD tool which supports AAA methodology. This free tool has been developed by the INRIA Rocquencourt France laboratory and several others laboratories contribute on its development over different research topics. AAA methodology aims at finding the best matching between an algorithm and an architecture while satisfying time constraints. SynDEx automatically generates a distributed and optimized synchronized executive.

Application algorithm is modeled by a data flow graph (DFG) to exhibit the potential parallelism between operations as represented by Figure 1. The algorithm model is a direct data dependence graph where each node models an operation and each oriented hyperedge models a data produced as output of a node and used as input of an other node or several others nodes. An operation is executed as soon as its input are available, and this DFG is infinitely repeated. SynDEx includes hierarchical algorithm representation, conditional statements (if...then...else) and iteration of algorithm parts (for...do...).

Architecture is also modeled by a graph, which is a directed graph where the vertices are operators (e.g microprocessors, DSP, FPGA) or media (e.g OCB busses, ethernet) and edges are connections between them. Operators have no internal parallelism computation available but the architecture exhibits the actual parallelism between operators. An example is shown Figure 2. In order to perform the adequation between these two graphs, operations and data dependencies have to be characterized (time execution) on each vertices of the architecture graph.

Adequation consists in performing a mapping and a scheduling of the operations and data transfers onto the operators and the communication media. It is carried out by a heuristic

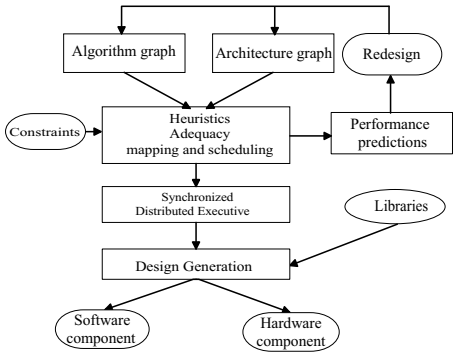


Fig. 3. SynDEx methodology flow

which takes into account durations of computations and inter-component communications. The result is a synchronized executive represented by a macro-code for each vertices of the architecture. Figure 3 depicts the overall methodology flow. This macro-code is composed of operations calls, communication interfaces, memory allocation directives, threads and semaphores for synchronization. Each macro-code is then translated toward a high level language for each HW/SW components. This translation produces an automatic dead-lock free code generation, macro-code directives are replaced by a corresponding code given in libraries (C/C++ for software components, VHDL for hardware components). Today this tool is used on heterogeneous architecture based on DSP and FPGA. Then we want to extend SynDEx capacities to runtime reconfigurable components.

III. Run time reconfiguration considerations for adequacy

A. Minimizing reconfiguration cost

Runtime partially reconfigurable components must be handled with a special processing during the adequation step. A major drawback of using runtime reconfiguration is the significant delay of hardware configuration. The total runtime of an application includes execution delay of each task on the hardware along with the total time spent for hardware reconfiguration between computations. The length of the sequence of reconfiguration is proportional to the reprogrammed area on the chip. Partial reconfiguration allows to change only a specified part of the design hence decreasing the reconfiguration time. An efficient way to minimize reconfiguration overhead is to overlap it as much as possible with the execution of others operations executed on others components. It is known as configuration prefetching [7]. Figure 4 illustrate our purpose. In this one we want to map and schedule an algorithm graph onto an architecture composed of two resources, one of them is dynamically reconfigurable. The algorithm, composed of six functions, is split into two branches. Function *D* needs the results of *C* and *Y* functions.

Operations $\{X, Y\}$ are assumed to be executed by the dynamic reconfigurable component of the architecture successively. Operations $\{A, B, C, D\}$ are implemented on the non-

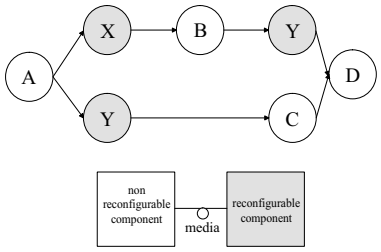


Fig. 4. Algorithm example

reconfigurable component. According to this algorithm, among all operations scheduling possibilities, one potential selection is:

S1:{A,X,B,Y,Y,C,D}

which is infinitively repeated. With this scheduling two reconfigurations are needed: one after computation of X to implement Y, one after the second computation of Y to implement X again. We have to define the following terms:

- C_k: *Computation cost of operation k.*
- $T_{j,k}$: *Cost of data transmission between operations j and k through the media.*
- R_k: *The reconfiguration delay of operation k.*
- $D_{j,k}$: *The time between the end of an operation j to the beginning of the operation k, both executed on the reconfigurable component.*
- P_k: *Prefetching cost of operation k.*

The prefetching cost of operation *k* is P_k = R_k-$D_{j,k}$, with j a previous different operation. So considering scheduling $S1$, we have to add the following prefetching cost to the first operation Y:

P_y = R_y-$D_{x,y}$
with $D_{x,y}$= $T_{x,b}$+C_b+$T_{b,y}$

In order to improve P_y, R_y must be similar to the delay between two reconfigurable operations. Hence a way to minimize prefetching overhead is to overlap a maximum of computations or communications with reconfigurations of the dynamic part. The heuristic of SynDEx has to be improved to take into account this overhead on reconfigurable operations.

B. Architecture graph modeling of runtime reconfigurable components

Runtime reconfigurable parts of an component must be considered as vertices in the architecture graph. As shown in the example in Figure 5, runtime reconfigurable parts of a FPGA (D1 and D2) and fixed parts (F1) can be represented as hardware operators of the architecture. An internal media (IL) allows data exchanges between them. (D1 and D2) will integrate the dynamic operator and the control to manage it.

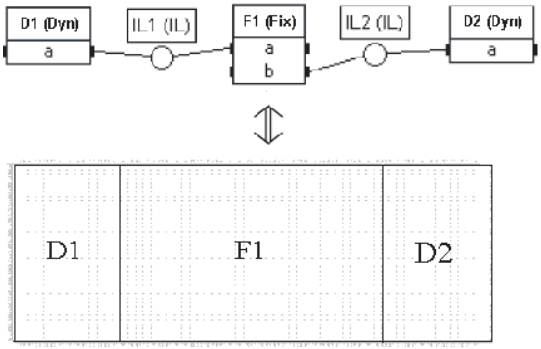


Fig. 5. Model of runtime reconfigurable parts of a FPGA with SynDEx

IV. Automatic design generation

A. General FPGA synthesis scheme

Once mapping and scheduling of the algorithm are performed, macro-code is automatically generated and each one must be translated. The translation generates the VHDL code, both for the static and dynamic parts of a FPGA. We use the GNU macro processor M4 and macros embedded in libraries. Different kinds of macro are developed for:

- communications,
- memory allocations,
- operator instantiations,
- synchronization using semaphores.

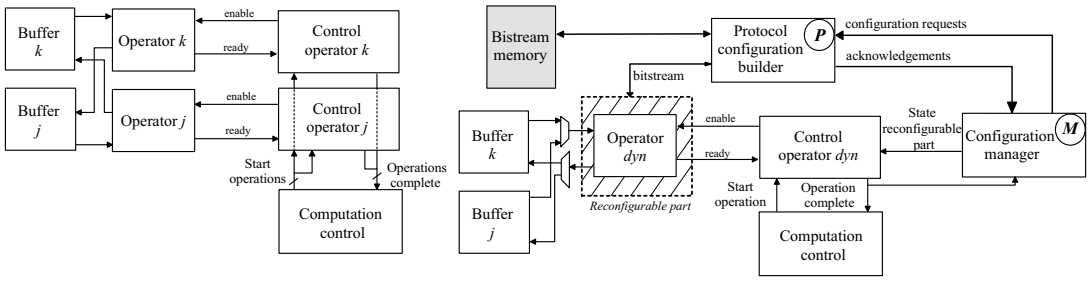
The final FPGA design is based on several processes:

- process to control communication sequencings,
- process to control computation sequencings,
- process which performs data transmission/reception,
- process for each kind of operator to control its behaviour,
- process which controls activation of reading and writing phases of buffers,

A same operator can be used at different time in the data flow, only one instantiation of each kind of operator is done in VHDL. We have defined an uniform interface for each operator through encapsulation. As described in the next section, we use global buffers which allow us to merge different kinds of buffer (depth and data width) filled by operators to store computation results. These two points lead to build complex data paths automatically. Our libraries are able to perform these constructions by using conditional VHDL signal assignment. In next section, we deal with two important points: optimization of memory for exchanged data and control of reconfigurations.

B. Macro-code preprocessing for memory minimization

The goal is to merge as much as possible independant buffers to minimize the total memory requirement, this is known as buffer merging technics [8]. We operate a macro-code preprocessing which analyzes data life of variables stored in these buffers and results in a list of buffers which



a) Non-reconfigurable architecture b) Dynamically reconfigurable architecture

Fig. 6. Architecture comparison between fixed/runtime reconfigurable solutions

must be merged into a global one. Buffers can be different as they must store computation results of operations working on various depth and data width (we consider only 8,16 or 32 bits width). Hence global buffers are automatically generated in agreement with operators data type computation results associated. We denote by:

- $L : \{b_1, b_2, ..b_n\}$: *A list of n buffers which can be merged, where* $L_{(k)} = b_k$.
- D_k : *The depth of buffer k.*
- W_k : *The data width of buffer k.*

Hence the total amount of memory needed is:

- Without buffer merging: $M_{\overline{bm}} = \sum_{i=1}^{n} D_{L(i)} * W_{L(i)}$
- With buffer merging: $M_{bm} = \max(D_{L(i)} * W_{L(i)})$ for $1 \leq i \leq n$.

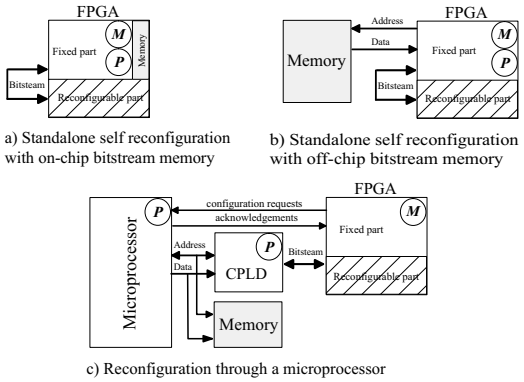
So the saved memory is:

$$S_{mem} = M_{\overline{bm}} - M_{bm}.$$

C. Design generation for runtime reconfigurable components

In order to perform reconfiguration of the dynamic part we have chosen to divide this process into two sub-parts: a configuration manager and a protocol configuration builder. Figure 6 shows a simple example based on two operations (*j* and *k*) which are executed successively. Case a) shows the design generated for a non-reconfigurable component, the two operators are physically implemented. Case *b)* is based on a dynamically reconfigurable component which implement successively the two operations. A configuration manager is in charge of the bitstream which must be loaded on the reconfigurable part by sending configuration requests. These requests are sent only when an operation has completed its computation and if a different operation has to be loaded after. So reconfigurations are performed as soon as the current operation is complete to enable configuration prefetching as described before. This functionality provides also information on the current state of the reconfigurable part, this is useful to start operator computations (with signal *'enable'*) only when the reconfiguration process is ended. Configuration requests are sent to the protocol configuration builder which is in charge to construct a valid reconfiguration stream in agreement with the protocol mode employed (e.g boundary scan). Encapsulation of operators with a standard interface allows to reconfigure only the area containing the operator without altering the design around. Functionalities involved in the general control of the dynamic area and the operator remain on a static part of the circuit with buffers. That allows to reduce the size of the bitstream which must be loaded and decrease the time needed to reconfigure.

Now this way to proceed must be adapted with architecture considerations. There are many ways to reconfigure partially a FPGA, Figure 7 shows three solutions of architectures for this purpose. Case a) shows a standalone self reconfiguration where the fixed part of the FPGA reconfigures the dynamic area. This case can be adapted for small amounts of bitstream data which can be stored by on-chip FPGA memory. However bitstreams which contain partial configurations require often a lot of memory and can't fit within the limited embedded memory provided by FPGAs, so bitstreams are stored by an external memory as depicted in the case b). The last case c), shows the used of a microprocessor to perform the reconfiguration. In this case the FPGA sends reconfiguration requests to the microprocessor through hardware interruptions for example. This microprocessor can be viewed as a slave for the FPGA. Either the microprocessor can act as a master by reconfiguring directly the dynamic area of the FPGA. The CPLD is used to interface these two components.



a) Standalone self reconfiguration with on-chip bitstream memory

b) Standalone self reconfiguration with off-chip bitstream memory

c) Reconfiguration through a microprocessor

Fig. 7. Different ways to reconfigure dynamic parts of a FPGA

Labels ***M*** and ***P*** show where are implemented functionalities *'Configuration manager'* and *'Protocol configuration builder'* respectively are implemented. Locations of these functionalities have a direct impact on the reconfiguration latency. Case c) has the highest reconfiguration latency since the 'protocol configuration builder' is a task of the microprocessor which can be activated through an hardware interruption. Moreover external memory accesses are often costing. Macro-codes generated for runtime reconfigurable components are handled by a special library. The *'Configuration manager'* is automatically generated in agreement with the sequencing of operations expressed in the macro-code. The reconfigurable part provides a virtual hardware, so at some time only one operator is physically implemented on this dynamic part.

Operations of the DFG implemented on this reconfigurable part are viewed as a global operation from the point of view of the control computation process. Computation invocations of these operations are renamed with a generic name in the macro-code. This renaming is feasible because there is no ambiguity on which operator is addressed when signal *'enable'* is driven (see figure 6). That allows to have only one control functionality for managing tasks implemented by the dynamically reconfigurable part.

V. Implementation example

Figure 8 shows an example based on a MC-CDMA transmitter. In this case the adequation algorithm - architecture is out of purpose. We want to map this application over an architecture composed of one DSP and one FPGA dynamically reconfigurable (Figure 9). Block *Spreading* is constrained to be executed on the dynamic part of the FPGA *(fpga_dyn)*. According to the spreading factor, two spreading operations (*fht4* or *fht8*), can be executed and selected by the entry condition *Cond*. Figure 10 represents a given mapping/scheduling performed by SynDEx for this application (reconfiguration times are not taken into account). Modulation process is mapped onto the DSP, interleaving and ifft onto the fixed part of the FPGA. According to entry *Cond* component Op_Dyn implements operations *fht4* or *fht8*.

A. Design generation

Figure 11 represents the computation sequencing for the partially reconfigurable part as expressed by the macro-code. We can merge buffers $\{B, C, D\}$ and operations $\{fht4, fht8\}$ are merged to generate operation *Op_Dyn*.

B. Implementation results

1) Xilinx design flow for partial reconfiguration: This example has been tested on a Virtex II FPGA from Xilinx. The code, both for fixed and dynamic part has been generated with SynDEx, through librairies. However, the generation of the bitstream needs a specific flow called modular design.

This design flow is broken down into three main phases: creating the floorplan and constraints for the overall design, implementing each module through the place and route process, assembling individual modules together into a complete

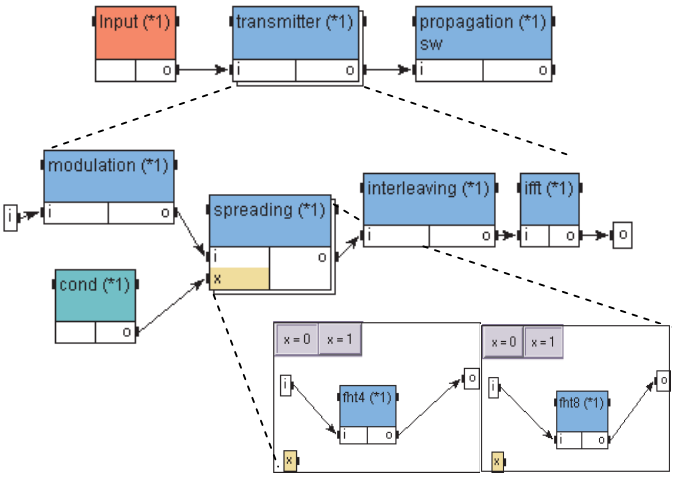


Fig. 8. Algorithm graph of a MC-CDMA transmitter

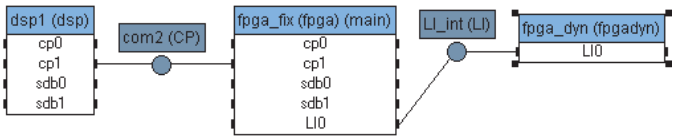


Fig. 9. Architecture graph

design. Module's boundary cannot be changed and the position and region occupied by any single reconfigurable module is always fixed. Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using a special bus macro. Bus macro implementation is based on 3-state buffers (TBUFs). The overall structure should be a top level design with each functional module defined as a "black-box" level of hierarchy. Hence as shown in the Figure 12 the operator *Op_Dyn* must be connected through the bus macro and instantiated in the top level of the FPGA design.

2) Numerical results of implementation: We have implemented the transmitter on a prototyping board from Sundance technology [9]. This board is composed of one DSP C6201 from Texas Instrument running at 200Mhz and one FPGA Xilinx Xc2v2000 partially reconfigurable. Virtex II has an internal reconfiguration access port (ICAP) based on a subset of the SelectMap interface. The previous example has been implemented according to two different ways corresponding to cases b) and c) of figure 7.

In case c), see figure 12, total or partial reconfigurations of the FPGA are performed through the bus CP by the DSP. Bitstreams are stored in the external memory of the DSP after they have been translated to an assembly file format. A part of the protocol configuration builder is a task of the DSP, this task is activated by hardware interruption and sends configuration data to the CPLD which configures the FPGA in mode SelectMap. Time to reconfigure the whole FPGA is nearly to 300 ms. On the other hand reconfiguration of the operator *Op_Dyn* takes about only 75 ms from the reconfiguration request to acknowledgment by the FPGA. This

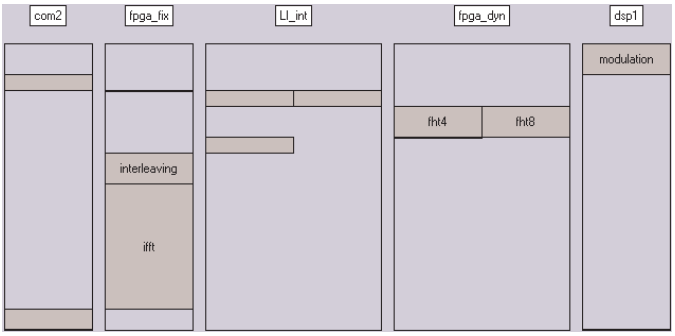


Fig. 10. Resulting scheduling over operators ans medias

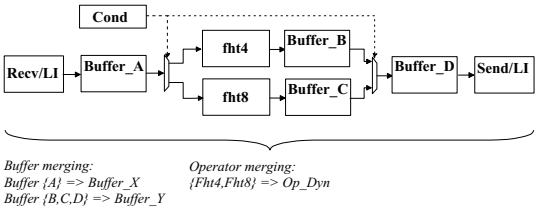


Fig. 11. Representation of macro-code computation sequencing for the partially reconfigurable part of the FPGA

represents 25% of the total configuration time.

For the case b) depicted in figure 13, at power-up the FPGA is fully configured with its bitstream stored in the external memory. Then partial reconfigurations are handle by the protocol configuration builder, which is in charge to address memory and drive ICAP. Reconfiguration is driven at 50Mhz and one bistream byte is loaded each cycle by the ICAP, hence reconfiguration of the operator *Op_Dyn* takes only 4.5 ms. So a self reconfiguration is much more efficient in time since there is no wasted DSP cycles in the reconfiguration process.

Nevertheless, in all cases bitstreams take a large amount of memory and need to be stored in external memory.

VI. Conclusion

We have described a methodology flow to manage automatically partially reconfigurable parts of a FPGA. It allows to map applications over heterogeneous architectures and fully exploit advantages given by partially reconfigurable components. The AAA methodology and associated tool SynDEx have been used to perform mapping and code generation for fixed and dynamic parts of FPGA. Either, SynDEx's heuristic needs additional developments to optimize time reconfiguration. Furthermore, complex design and architecture can support more than one dynamic parts.

This design flow has the main advantage to target as well as software components as hardware components to implement complex applications from a high level functional description.

We plan to apply this design flow for radio system telecommunication architectures and implement various telecommunication standards over a reconfigurable platform.

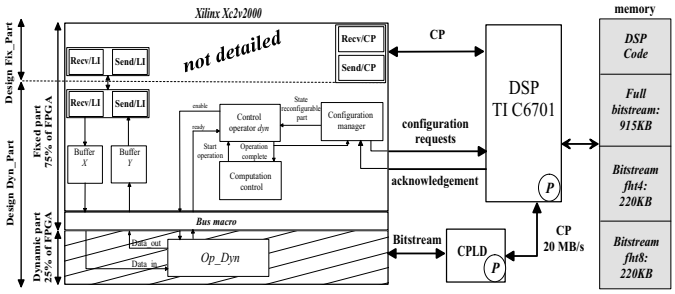


Fig. 12. Implementation architecture and internal FPGA dynamic part design detailed - DSP reconfiguration based -

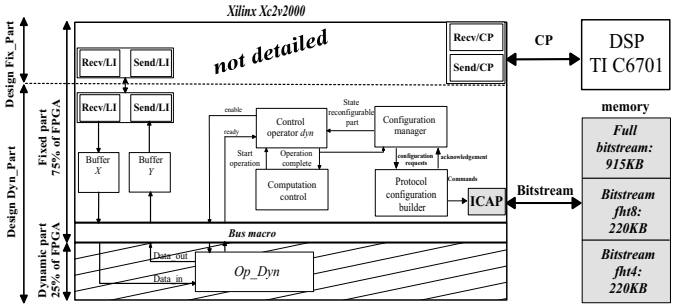


Fig. 13. Implementation architecture and internal FPGA dynamic part design detailed - self reconfiguration based -

References

[1] Mitola J., "Software radio architecture evolution: Foundations, technology tradeoffs, and architecture implications," *IEICE Trans. Commn.*, vol. E83-B, no. 6, pp. 1165–1172, June 2000.

[2] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour, "Dynamic hardware plugins in an fpga with partial runtime reconfiguration," *Design Automation Conference (DAC)*, 2002.

[3] J. Noguera and R. Badia, "A hw/sw partitioning algorithm for dynamically reconfigurable architectures," *Proc. Design Automation and Test in Europe*, vol. pp. 72934, 200t, 2001.

[4] Y. Lavarenne C. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," *Formal Methods and Models for Codesign Conference, France*, June 2003.

[5] T.M.McGinnity J.Harkin and L.P.Maguire, "Partitioning methodology for dynamically reconfigurable embedded systems," *IEE Proc-Comput. Digit. Tech*, vol. 147, November 2000.

[6] Sebastien Lenours, "Etude, optimisation et implementation de systemes mc-cdma sur architectures heterogenes," *PHD Thesis*, vol. D03 - 17, Decembre 2003.

[7] S HAUCK, "Configuration prefetch for single context reconfigurable coprocessors," *ACM/SIGDA International Symposium on FPGAs*, vol. E83-B, no. 6, pp. 6574, June 1998.

[8] E. A. Lee P. K. Murthy, S. S. Bhattacharyya, "Minimizing memory requirements for chain-structured sdf graphs," *Proc. of ICASSP, Australia*, vol. E83-B, no. 6, pp. 6574, June 1994.

[9] Sundance Multiprocessor Technology Ltd, "http://www.sundance.com," .